METHODS AND APPARATUS FOR PROCESSOR TASK MIGRATION IN A MULTI-PROCESSOR SYSTEM

BACKGROUND OF THE INVENTION

[0001]   The present invention relates to methods and apparatus for managing processor tasks in a multi-processing system and, in particular, for scheduling and executing the processor tasks among the sub-processing units of the multi-processing system on a substantially self governing basis.

[0002]   Real-time, multimedia applications are becoming increasingly important. These applications require extremely fast processing speeds, such as many thousands of megabits of data per second. While single processing units are capable of fast processing speeds, they cannot generally match the processing speeds of multi-processor architectures. Indeed, in multi-processor systems, a plurality of sub-processors can operate in parallel (or at least in concert) to achieve desired processing results.

[0003]   The types of computers and computing devices that may employ multi-processing techniques are extensive. In addition to personal computers (PCs) and servers, these computing devices include cellular telephones, mobile computers, personal digital assistants (PDAs), set top boxes, digital televisions and many others.

[0004]   Real-time, multimedia software applications are constructed of processing code, such as processing instructions and data. A collection of at least some of the processing instructions and/or the processing data may be referred to as a processor task. While the program statements within a processor task may be executed sequentially, different processor tasks may be executed in parallel on different processors in a multi-processor system. Thus, the

software applications may be considered to contain processor tasks to be executed by the multi-processing system.

[0005]   A design concern in a multi-processing system is how to manage which sub-processing units of the system execute which processor tasks.   In some multi-processing systems, the processor tasks specify which of the sub-processing units are to perform the execution.   A drawback to this approach is that the programmer may not have optimized the allocation of processor tasks among the sub-processing units.   For example, one   or   more   processor   tasks   may   specify   the   same sub-processing unit at the same time.   This dictates that some of the processor tasks be put on hold until the specified sub-processing unit becomes available, thereby delaying their execution.   Unfortunately   this   also   results   in   an unpredictable latency as to the execution of the processor tasks.

[0006]   Other systems contemplate that a managing element communicates with the sub-processing units and schedules the processor tasks among them.   A communication protocol must therefore be in place to facilitate such communication. Unfortunately communication protocols often result in message delays between the managing element and the sub-processing units.   Indeed, such protocols may require the use of a memory mapped I/O space (using memory mapped registers), which are generally slow.   Further, the managing elements (which may themselves be processors of the system) may employ multiple partition domains, which may require significant time to change (e.g., 700 us).   These characteristics also delay execution of the processor tasks and result in unpredictable latencies.   Thus, overall processor throughput and efficiency of the multi-processing system are sacrificed, which may result in significant impact on the real-time and/or multimedia experience of a user of the system.

[0007]   Therefore, there are needs in the art for new methods and apparatus for achieving efficient multi-processing that reduces the adverse affects of hard processor errors.

SUMMARY OF THE INVENTION

[0008]   In accordance with one or more aspects of the present invention, a method of executing processor tasks on a multi-processing system is contemplated. The multi-processing system includes a plurality of sub-processing units and a main processing unit that may access a shared memory. Each sub-processing unit includes an on-chip local memory separate from the shared memory. The method includes: providing that the processor tasks be copied from the shared memory into the local memory of the sub-processing units in order to execute them, and prohibiting the execution of the processor tasks from the shared memory; and migrating at least one processor task from one of the sub-processing units to another of the sub-processing units.

[0009]   The migration of the at least one processor task is preferably based on a condition. The condition may be based on respective priority levels associated with the processor tasks.

[0010]   In one or more embodiments of the invention, satisfaction of the condition and the initiating of the migration is not based on preemptive action. Preferably, the method further includes requiring that the sub-processing units select processor tasks from the shared memory for execution based on their priority levels. The method may also include: requiring that the sub-processing units select a processor task of higher priority before a processor task of lower priority from the shared memory.

[0011]   Preferably, the method further includes selecting a first processor task of a first priority level from the shared memory for execution by a first sub-processing unit; selecting

a second processor task of a second priority level from the shared memory for execution by a second sub-processing unit; and yielding the first sub-processing unit to a third processor task of a third priority level, third processor task being selected because its priority level is higher than any other processor tasks that a ready to be executed.

[0012] The method may also include: writing the first processor task from the local memory of the first sub-processing unit back to the shared memory. Preferably, the method includes: yielding the second sub-processing unit to the first processor task, the first processor task being selected because its priority level is higher than any other processor tasks in the shared memory that are ready to be executed, thereby non-preemptively migrating the first processor task from the first sub-processing unit to the second sub-processing unit. The method may also include: writing the second processor task from the local memory of the second sub-processing unit back to the shared memory.

[0013] The method also contemplates that a processor task of lower priority running on one of the sub-processing units may be preemptively replaced with a processor task of higher priority. The method may also include: requiring that the sub-processing units select a processor task of higher priority before a processor task of lower priority from the shared memory.

[0014] Preferably, the method further includes: selecting a plurality of processor tasks of associated priority levels from the shared memory for execution by a number of sub-processing units; causing an n-th processor task in the shared memory having a given priority level to become ready for execution; and determining whether the given priority level is higher than any of the priority levels of the plurality of processor tasks. Preferably, at least one of the sub-processing units is operable to perform the determination.

The method may also include preemptively replacing one of the plurality of processor tasks of lower priority level than the given priority level with the n-th processor task.

[0015] One or more of the sub-processing units may be operable to at least initiate the replacement and cause the one of the plurality of sub-processing units to yield execution of the processor task of lower priority level. The method may include: providing that the initiating sub-processing unit issues an interrupt to the yielding sub-processing unit in order to initiate the replacement of the processor task of lower priority level. Preferably, the method also includes providing that the yielding sub-processing unit writes the processor task of lower priority from its local memory back into the shared memory.

[0016] In accordance with one or more further aspects of the present invention, the method preferably includes providing that the sub-processing units may determine whether an n-th processor task in the shared memory having a given priority level is higher than any of the priority levels of the plurality of processor tasks.

[0017] Preferably, the sub-processing units use a shared task priority table in determining whether the n-th processor task is of a higher priority level than the plurality of processor tasks. The shared task priority table may include entries of for sub-processing unit identifiers and processor task priority identifiers; and each entry may include a sub-processing unit identifier and priority identifier pair that indicate a priority level of a given processor task running on an associated sub-processing unit.

[0018] A sub-processing unit seeking to determine whether the n-th processor task is of a higher priority level than the plurality of processor tasks may search the shared task priority table to find an entry pair indicating a lower priority level. Preferably, the lower priority level is the

lowest priority level of the shared task priority table. The sub-processing units preferably modify the shared task priority table such that the entry pairs are current.

[0019] In accordance with one or more further aspects of the present invention, a method includes: providing that the processor tasks be copied from the shared memory into the local memory of the sub-processing units in order to execute them, and prohibiting the execution of the processor tasks from the shared memory; providing that the sub-processing units select processor tasks from the shared memory for execution based on priority levels of the processor tasks; and migrating a processor task of higher priority running on a given one of the sub-processing units to another of the sub-processing units running a processor task of lower priority in response to an interrupt received by the given sub-processing unit.

[0020] The method may also include: selecting a plurality of processor tasks of associated priority levels from the shared memory for execution by a number of sub-processing units; receiving the interrupt at the given sub-processing unit; and determining which of the plurality of processor tasks running on the number of sub-processing units has a lowest priority level that is lower than the priority level of the processor task running on the given sub-processing unit. The given sub-processing unit may be operable to perform the determination.

[0021] The method preferably further includes: migrating the given processor task to the sub-processing unit running the processor task of lowest priority level and replacing that processor task. The given sub-processing unit may be operable to at least initiate the migration and causing the sub-processing unit running the processor task of lowest priority level to yield execution to the given processor task of higher priority level. The given sub-processing unit

preferably issues an interrupt to the yielding sub-processing unit in order to initiate the replacement of the processor task of lowest priority level.

[0022] The yielding sub-processing unit may write the processor task of lower priority from its local memory back into the shared memory. The yielding sub-processing unit may also copy the given processor task of higher priority from the local memory of the given sub-processing unit into its local memory for execution.

[0023] The given sub-processing unit preferably uses a shared task priority table in determining which processor task is of the lowest priority level. The shared task priority table may include entries of for sub-processing unit identifiers and processor task priority identifiers; and each entry may include a sub-processing unit identifier and priority identifier pair that indicate a priority level of a given processor task running on an associated sub-processing unit. The given sub-processing unit may search the shared task priority table to find an entry pair indicating a lowest priority level. The sub-processing units preferably modify the shared task priority table such that the entry pairs are current.

[0024] In accordance with one or more further aspects of the present invention, a multi-processor apparatus includes: a plurality of sub-processing units, each sub-processing unit including an on-chip local memory in which to execute processor tasks; and a shared memory operable to store processor tasks that are ready to be executed. The processor tasks are copied from the shared memory into the local memory of the sub-processing units in order to execute them, and the processor tasks are not executed from the shared memory, and at least one processor task is migrated from one of the sub-processing units to another of the sub-processing units.

[0025]   In accordance with one or more further aspects of the present invention, the sub-processing units of the apparatus may be operable to perform the method steps as discussed above.

[0026]   Other aspects, features, and advantages of the present invention will be apparent to one skilled in the art from the description herein taken in conjunction with the accompanying drawings.

DESCRIPTION OF THE DRAWINGS

[0027]   For the purposes of illustration, there are forms shown in the drawings that are presently preferred, it being understood, however, that the invention is not limited to the precise arrangements and instrumentalities shown.

[0028]   FIG. 1 is a diagram illustrating the structure of a multi-processing system in accordance with one or more aspects of the present invention;

[0029]   FIG. 2 is a block diagram illustrating the storage of processor tasks in a shared memory;

[0030]   FIG. 3 is a diagram illustrating a preferred structure of a processor element (PE) in accordance with the present invention;

[0031]   FIG. 4 is a diagram illustrating the structure of an exemplary sub-processing unit (SPU) in accordance with the present invention;

[0032]   FIG. 5 is an illustration of an example of a processor task table that may be used in accordance with one or more aspects of the present invention;

[0033]   FIG. 6 is a state diagram of a linked list of processor tasks established by the task table of FIG. 5;

[0034]   FIG. 7 is an illustration of an example of a task queue that may be used in conjunction with the task table of FIG. 5 to manage the execution of processor tasks;

[0035]   FIG. 8 is a flow diagram illustrating process steps that may be carried out by the multi-processing system in accordance with one or more aspects of the present invention;

[0036]   FIG. 9 is a flow diagram illustrating further process steps that may be carried out by the multi-processing system in accordance with the present invention;

[0037]   FIG. 10 is a flow diagram illustrating still further process steps that may be carried out by the multi-processing system in accordance with the present invention;

[0038]   FIG. 11 is a flow diagram illustrating process steps for initializing the processor tasks in the shared memory and that may be carried out by the multi-processing system in accordance with various aspects of the present invention;

[0039]   FIG. 12 is a state diagram illustrating the different status states of the processor tasks in accordance with one or more aspects of the present invention;

[0040]   FIG. 13 is a block diagram illustrating how processor tasks may be copied from and written back to the shared memory in accordance with one or more aspects of the present invention;

[0041]   FIG. 14 is a timing diagram illustrating the processing latency associated with the copy and write back technique of FIG. 13;

[0042]   FIG. 15 is a block diagram illustrating how processor tasks may be copied from and written back to the shared memory in accordance with one or more further aspects of the present invention;

[0043]   FIG. 16 is a timing diagram illustrating the processing latency associated with the copy and write back technique of FIG. 15;

[0044]   FIG. 17 is a block diagram illustrating how processor tasks may be copied from and written back to the shared memory in accordance with one or more still further aspects of the present invention;

[0045] FIG. 18 is a timing diagram illustrating the processing latency associated with the copy and write back technique of FIG. 17;

[0046] FIG. 19 is a block diagram illustrating how processor tasks may be copied from and written back to the shared memory in accordance with one or more still further aspects of the present invention;

[0047] FIG. 20 is a block diagram illustrating non-preemptive processor task migration features of certain aspects of the present invention;

[0048] FIG. 21 is a block diagram illustrating further non-preemptive processor task migration features of certain aspects of the present invention;

[0049] FIG. 22 is a block diagram illustrating still further non-preemptive processor task migration features of certain aspects of the present invention;

[0050] FIG. 23 is a block diagram illustrating preemptive multi-tasking features of certain aspects of the present invention;

[0051] FIG. 24 is a block diagram illustrating further preemptive multi-tasking features of certain aspects of the present invention;

[0052] FIG. 25 is a block diagram illustrating preemptive processor task migration features of certain aspects of the present invention;

[0053] FIG. 26 is a block diagram illustrating further preemptive processor task migration features of certain aspects of the present invention;

[0054] FIG. 27 is a partial block diagram and partial flow diagram illustrating certain processor interrupt techniques in accordance with one or more aspects of the present invention;

[0055] FIG. 28 is a partial block diagram and partial flow diagram illustrating further processor interrupt techniques in accordance with one or more aspects of the present invention;

[0056] FIG. 29 is a partial block diagram and partial flow diagram illustrating still further processor interrupt techniques in accordance with one or more aspects of the present invention;

[0057] FIG. 30 is a diagram illustrating the structure of a processing system including more than one sub-processing unit in accordance with one or more further aspects of the present invention;

[0058] FIG. 31 is a system diagram of a distributed multi-processing system in accordance with one or more aspects of the present invention; and

[0059] FIG. 32 is a block diagram of a software cell feature that may be used in conjunction with the multi-processing units of the present invention.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENTS

[0060] With reference to the drawings, where like numerals indicate like elements, there is shown in FIG. 1 a multi-processing system 100 in accordance with one or more aspects of the present invention. The multi-processing system 100 includes a plurality of processors 102 (any number may be used) coupled to a shared memory 106, such as a DRAM, over a bus 108. It is noted that the shared memory 106 need not be a DRAM; indeed, it may be formed using any known or hereinafter developed technology.

[0061] One of the processors 102 is preferably a main processing unit, for example, processing unit 102A. The other processing units 102 are preferably sub-processing units (SPUs), such as processing units 102B, 102C, 102D, etc. The sub-processing units 102 may be implemented using any of the known or hereinafter developed computer architectures. All of the sub-processing units 102 need not be implemented using the same architecture; indeed they may be of heterogeneous or homogenous configurations. It is noted that the main

-11-

processing unit 102A may be disposed locally with respect to the sub-processing units 102B-D, such as in the same chip, in the same package, on the same circuit board, in the same product, etc. Alternatively, the main processing unit 102A may be remotely located from the sub-processing units 102B-D, such as in different products, which may be coupled over a bus, a communications network (such as the Internet) or the like. Similarly, the sub-processing units 102B-D may be locally or remotely located from one another.

[0062] The main processing unit 102A may be used to schedule and orchestrate the processing of data and applications by the sub-processing units 102B-D such that the sub-processing units 102B-D perform the processing of these data and applications in a parallel and independent manner. In accordance with some aspects of the present invention, however, the main processing unit 102A does not take a central role in scheduling the execution of processor tasks among the sub-processing units. Rather, such scheduling is preferably left to the SPUs themselves.

[0063] The assignment of roles and functions to the processors 102 in FIG. 1 is flexible. For example, any of the processors 102 can be a main processor unit or sub-processing units.

[0064] With reference to FIG. 2, the main processing unit 102A preferably assumes the role of a service processor to the SPUs 102B-F, particularly when it comes to the scheduling and management of the processor tasks among the SPUs. In accordance with some aspects of the present invention, the main processing unit 102A may evaluate the processor tasks that are contained within a software application and take part in the allocation of the shared memory 106, the allocation of the SPUs, and the initial storage of the processor tasks 110 within the shared memory 106. As to the allocation of the shared memory 106, the main processing unit 102A preferably

-12-

determines how much of the memory space should be allocated to a given number of processor tasks 110. In this regard, the main processing unit 102A may allocate a first area 106A of the shared memory 106 for storage of some processor tasks 110, and may allocate a second area 106B of the shared memory 106 for storage of other processor tasks 110. The main processing unit 102A may also establish rules as to data synchronization within the respective areas 106A and 106B of the shared memory 106.

[0065] In accordance with one or more further aspects of the invention, the respective areas 106A and 106B of the shared memory 106 can only be accessed by a prescribed number of sub-processing units 102, such as the sub-processing units 102 that are allocated to perform the particular processor tasks 110 that are stored within a given area of the shared memory 106. For example, sub-processing units 102B-D are preferably the only SPUs that are permitted to access the processor tasks 110 within the first area 106A of the shared memory 106. Likewise, the sub-processing units 102E-F are preferably the only SPUs that are permitted to access the processor tasks 110 within the second area 106B of the shared memory 106. Further details concerning the techniques of protecting the respective areas 106A and 106B of the shared memory 106 may be found in U.S. Patent No. 6,526,491, entitled MEMORY PROTECTION SYSTEM AND METHOD FOR COMPUTER ARCHITECTURE FOR BROADBAND NETWORKS, the entire disclosure of which is hereby incorporated by reference.

[0066] In accordance with one or more further aspects of the present invention, once the processor tasks 110 are disposed within the shared memory 106 and the sub-processing units 102 are allocated to execute the tasks, the main processing unit 102A preferably does not participate in the scheduling and management of the execution of the processor

tasks 110. Instead those responsibilities are left to the particular sub-processing units 102 involved.

[0067] Prior to discussing further details regarding the processor task management features of the various embodiments of the present invention, a description of a preferred computer architecture for implementing a multi-processor system will be discussed. In this regard, reference is now made to FIG. 3, which is a block diagram of a basic processing module or processor element (PE) 200. In accordance with this computer architecture, all sub-processors of a multi-processing computer system are constructed from a common computing module (or cell). This common computing module has a consistent structure and preferably employs the same instruction set architecture. In alternative embodiments of the present invention, the sub-processing units may be of heterogeneous configurations. The multi-processing computer system can be formed of one or more clients, servers, PCs, mobile computers, game machines, PDAs, set top boxes, appliances, digital televisions and other devices using computer processors.

[0068] The basic processing module is a processor element (PE). As shown in FIG. 3, the PE 200 comprises an I/O interface 202, a processing unit (PU) 204, a direct memory access controller (DMAC) 206, and a plurality of sub-processing units 208, namely, sub-processing unit 208A, sub-processing unit 208B, sub-processing unit 208C, and sub-processing unit 208D. A local (or internal) PE bus 212 transmits data and applications among the PU 204, the sub-processing units 208, the DMAC 206, and a memory interface 210. The local PE bus 212 can have, e.g., a conventional architecture or can be implemented as a packet switch network. Implementation as a packet switch network, while requiring more hardware, increases available bandwidth.

[0069]   The PE 200 can be constructed using various methods
for implementing digital logic.   The PE 200 preferably is
constructed, however, as a single integrated circuit employing
a complementary metal oxide semiconductor (CMOS) on a silicon
substrate.   Alternative   materials   for   substrates   include
gallium   arsinide,   gallium   aluminum   arsinide   and   other
so-called III-B compounds employing a wide variety of dopants.
The PE 200 also could be implemented using superconducting
material, e.g., rapid single-flux-quantum (RSFQ) logic.

[0070]   The PE 200 is closely associated with a dynamic
random access memory (DRAM) 214 through a high bandwidth
memory connection 216.   The DRAM 214 functions as the main (or
shared)   memory   for   the   PE   200.   Although   the   DRAM   214
preferably is a dynamic random access memory, the DRAM 214
could be implemented using other means, e.g., as a static
random access memory (SRAM), a magnetic random access memory
(MRAM), an optical memory, a holographic memory, etc.   The
DMAC 206 and the memory interface 210 facilitate the transfer
of data between the DRAM 214 and the sub-processing units 208
and the PU 204 of the PE 200.   It is noted that the DMAC 206
and/or   the   memory   interface   210   may   be   integrally   or
separately disposed with respect to the sub-processing units
208   and   the   PU   204.   Indeed,   instead   of   a   separate
configuration as shown, the DMAC 206 function and/or the
memory interface 210 function may be integral with one or more
(preferably all) of the sub-processing units 208 and the PU
204.

[0071]   The   PU   204   can   be,   e.g.,   a   standard   processor
capable of stand-alone processing of data and applications.
The   sub-processing   units   208   preferably   are   single
instruction,   multiple   data   (SIMD)   processors.   The
sub-processing units 208 perform the processing of data and
applications, preferably in a parallel and independent manner.
The   DMAC   206   controls   accesses   by   the   PU   204   and   the

sub-processing units 208 to the data and applications (e.g., the processor tasks 110) stored in the shared DRAM 214. It is noted that the PU 204 may be implemented by one of the sub-processing units 208 taking on the role of a main processing unit.

[0072]   In accordance with this modular structure, the number of PEs 200 employed by a particular computer system is based upon the processing power required by that system. For example, a server may employ four PEs 200, a workstation may employ two PEs 200 and a PDA may employ one PE 200. The number of sub-processing units of a PE 200 assigned to processing a particular software cell depends upon the complexity and magnitude of the programs and data within the cell.

[0073]   FIG. 4 illustrates the preferred structure and function of a sub-processing unit 208. The sub-processing unit 208 includes local memory 250, registers 252, one or more floating point units 254 and one or more integer units 256. Again, however, depending upon the processing power required, a greater or lesser number of floating points units 254 and integer units 256 may be employed. The floating point units 254 preferably operate at a speed of 32 billion floating point operations per second (32 GFLOPS), and the integer units 256 preferably operate at a speed of 32 billion operations per second (32 GOPS).

[0074]   In a preferred embodiment, the local memory 250 contains 256 kilobytes of storage, and the capacity of registers 252 is 128 X 128 bits. It is noted that the processor tasks 110 are not executed using the shared memory 214. Rather, the tasks 110 are copied into the local memory 250 of a given sub-processing unit 208 and executed locally.

[0075]   The local memory 250 may or may not be a cache memory. The local memory 250 is preferably constructed as a static random access memory (SRAM). A PU 204 may require

-16-

cache coherency support for direct memory accesses initiated by the PU 204. Cache coherency support is not required, however, for direct memory accesses initiated by the sub-processing units 208 or for accesses from and to external devices.

[0076] The sub-processing unit 208 further includes a bus interface (I/F) 258 for transmitting applications and data to and from the sub-processing unit 208. In a preferred embodiment, the bus I/F 258 is coupled to a DMAC 206, which is shown in dashed line to indicate that it may be integrally disposed within the sub-processing unit 208 as shown or may be externally disposed (as shown in FIG. 3). A pair of busses 268A, 268B interconnect the DMAC 206 between the bus I/F 258 and the local memory 250. The busses 268A, 268B are preferably 256 bits wide.

[0077] The sub-processing unit 208 further includes internal busses 260, 262 and 264. In a preferred embodiment, the bus 260 has a width of 256 bits and provides communications between the local memory 250 and the registers 252. The busses 262 and 264 provide communications between, respectively, the registers 252 and the floating point units 254, and the registers 252 and the integer units 256. In a preferred embodiment, the width of the busses 264 and 262 from the registers 252 to the floating point or the integer units is 384 bits, and the width of the busses 264 and 262 from the floating point or the integer units 254, 256 to the registers 252 is 128 bits. The larger width of these busses from the registers 252 to the floating point or the integer units 254, 256 than from these units to the registers 252 accommodates the larger data flow from the registers 252 during processing. A maximum of three words are needed for each calculation. The result of each calculation, however, normally is only one word.

[0078]   Turning again to the various processor tasks management features of the present invention, and with reference to FIG. 2, it is preferable that the sub-processing units 102 utilize a task table in order to determine which of the processor tasks 110 should be copied from the shared memory 106 and into one of the local memories of the SPUs 102 for execution.   In this regard, reference is now made to FIG. 5, which is a conceptual illustration of a task table 280 that may be utilized in accordance with various aspects of the present invention. The task table 280 is preferably stored in the shared memory 106 (the details of how the task table 280 is initialized will be discussed later).   The task table 280 preferably includes a plurality of task table entries T1, T2, T3, etc.   Each task table entry is preferably associated with one of the processor tasks 110 (FIG. 2), for example, by way of associative addressing or some other means for relating the task table entries to the processor tasks 110.

[0079]   In a preferred embodiment, each of the task table entries may include at least one of a status indication (STAT), a priority indication (PRI), and a pair of pointers (PREV, NEXT). The STAT preferably provides an indication as to whether the processor task associated with the given task table entry is READY to be executed by (or is RUNNING on) one or more of the sub-processing units.   The PRI preferably provides an indication as to a priority level of the associated processor task 110.   There may be any number of priority levels associated with the processor tasks 110, which may be established by the software programmer or may be later established through execution of the software application.   In any event, the priority levels of the processor tasks 110 may be utilized to establish an order in which the processor tasks are executed.   The PREV value is preferably a pointer to a previous task table entry (or to a previous processor task 110) in an ordered list of linked task table entries (or list

-18-

of processor tasks). The NEXT value is preferably a pointer to a NEXT task table entry (or processor task) in the ordered list of linked task table entries (or processor tasks).

[0080] In accordance with one or more aspects of the present invention, the task table 280 is preferably utilized by the sub-processing unit 102 in order to determine an order in which the processor tasks 110 are copied from the shared memory 106 for execution. For example, in order to properly execute a software application on the multi-processing system 100 or 200, certain processor tasks 110 may need to be executed in a particular order, or at last in terms of a general order, say T1, T8, T6, and T9. In order to reflect this illustrative example of processor task ordering, the task table 280 preferably contains pointers in the respective PREV and NEXT portions of the task table entries that create a linked list of task table entries, and by extension processor tasks. In accordance with the particularities of the above example, the task table entry T1 includes a NEXT value that points to task table entry T8. Task table entry T8 includes a PREV value that points to task table entry T1, and contains a NEXT value that points to task table entry T6. Task table entry T6 includes a PREV value that points to task table entry T8, and a NEXT value that points to task table entry T9. Task table entry T9 includes a PREV value that points to task table entry T6.

[0081] With reference to FIG. 6, the linked list of task table entries of the above example may be conceptually illustrated as a state diagram in which a transition from a particular processor task associated with task table entry T1 results in the selection and execution of another processor task that is associated with task table entry T8. A transition from the processor task associated with task table entry T8 results in the selection and execution of a processor task associated with task table entry T6, and so on. A

circular association of the task table entries (and/or the processor tasks themselves) may be obtained by insuring that the first, or head, task table entry T1 includes a PREV value that points to task table entry T9, and that task table entry T9 includes a NEXT value that points to task table entry T1.

[0082] During operation, the respective sub-processing units 102 that are allocated to execute the pool of processor tasks 110 (preferably within a given area 106A or 106B) of the shared memory 106 first access the task table 280 to determine which processor task 110 should be taken up next for execution. In order to aid in the identification of the first (or head) entry of the linked list, the sub-processing units 102 preferably also have access to a task queue 282, which is conceptually illustrated in FIG. 7. The task queue 282 preferably includes an entry for each priority level of the associated processor tasks 110. Each entry preferably includes at least one of a HEAD pointer and a TAIL pointer.

[0083] With further reference to FIG. 6, the illustrated state diagram of the exemplary linked list is representative of processor tasks 110 having a priority level of 1. Indeed, each task table entry (FIG. 5) for entries T1, T8, T6, and T9 include PRI values of 1.

[0084] The HEAD pointer and TAIL pointer of the task queue entry associated with priority level 1 contain pointers to task table entry T1 and task table entry T9, respectively. The other entries of the task queue 282 are associated with the HEAD and TAIL pointers of other priority levels for other linked lists. In this manner, it is contemplated that various embodiments of the present invention may include multiple linked lists of task table entries (and by extension processor tasks), where each linked list contains entries of the same or a least similar priority levels. The respective sub-processing units 102 preferably utilize the task table 280 and the task queue 282 to determine which of the processor

tasks 110 should be copied from the shared memory 106 for execution. Provided that the respective linked lists are created and maintained properly, the processor tasks 110 may be executed in the proper order in order to achieve desirable results in the execution of the overall software application.

[0085] In accordance with various aspects of the present invention, the sub-processing units 102 maintain and modify the task table 280 and the task queue 282 during execution of the software application. In this regard, reference is now made to FIGS. 8-10, which are flow diagrams illustrating a process flow that is suitable for achieving one or more desirable features of the present invention. At action 300, a particular sub-processing unit 102 is called to initiate the copying of a processor task 110 from the shared memory 106 to the local memory thereof. At action 302, the sub-processing unit 102 locks and copies the task queue 282 into its local memory. Thereafter, the task queue 282 is searched for a highest priority ready task (action 304). Using the example illustrated in FIG. 7, the task queue 282 includes a HEAD pointer that points to the task table entry T1, which is associated with a processor task of highest priority, such as priority level 1. Since the processor task associated with task table entry T1 is targeted for execution, the sub-processing unit 102 preferably modifies the task queue 282 to remove the reference to that processor task (action 306). In accordance with a preferred embodiment, this entails modification of the HEAD pointer to the TASK table entry T1 to another task table entry that will become a new first (or head) task table entry indicating the next processor task that should be taken up for execution. In particular, the NEXT pointer of the task table entry T1 may be used as the new HEAD pointer of priority level 1. Indeed, as illustrated in FIG. 6, once the processor task associated with task table entry T1 is being executed (or running) it is no longer in the

READY state and should be removed from the state diagram. This should leave the task table entry T8 as the head entry of the state diagram. As task table entry T1 will no longer be part of the READY state diagram, the PREV pointer of the task table entry T8 may be modified to point to task table entry T9. Thus, at action 308 that task table is labeled and copied into the local memory of the SPU 102 such that it may be modified. Similarly, the NEXT pointer of the task table entry T9 may be modified to point to the task table entry T8.

[0086]   In accordance with preferable aspects of the present invention, the SPU 102 preferably modifies the STAT value of the task table entry T1 from READY to RUNNING (action 310, FIG. 9). As action 312, a determination is preferably made as to whether the SPU 102 was running a previous task at the time that it was called (action 300) to invoke a next task. This may occur when a previous task running on the SPU 102 yields to another task. For purposes of the present example, it is assumed that a previous task neither yielded to, nor was otherwise usurped by, the next processor task 110 and that the result of the determination at action 312 is in the negative. Accordingly, the process flow preferably advances to action 318, where the SPU 102 writes the modified task queue 282 and the modified task table 280 back into the shared memory 106. At this point, the task table 280 and the task queue 282 have been updated and, in accordance with preferred synchronization techniques, are unlocked such that they may be copied and modified by other sub-processing units 102.

[0087]   If the result of the determination at action 312 is in the affirmative, such as would be the case if a previous processor task 110 yielded to the next processor task for execution, the process flow preferably advances to action 314. There, the SPU preferably modifies the STAT value of the task table entry associated with the yielding processor task from RUNNING to READY. Further, the SPU may then modify the PREV

-22-

and NEXT pointers of various task table entries (including the task table entry associated with the yielding processor task) in order to reintroduce the yielding processor task back into the appropriate linked list. Preferably, this is accomplished by referencing the priority level of the yielding processor task 110 as reflected in the PRI value of the associated task table entry. At action 316, the yielding processor task may be written back into the shared memory 106 such that it may be taken up at a later time. Thereafter, the process flow advances to action 318, where the task queue 282 and the task table 280 are written back into the shared memory 106.

[0088] At action 320 (FIG. 10), the next processor task 110 (e.g., the processor task associated with task table entry T8) is copied by the sub-processing unit 102 from the shared memory 106 to the local memory thereof. At action 322, the sub-processing unit 120 preferably restores and/or updates registers thereof (e.g., with any data associated with the new processor task) for use in executing the new processor task 110. Finally, at action 324, the new processor task 110 is executed by the sub-processing unit 102.

[0089] It is noted that the above sequence of actions is presented by way of example only and it should be understood by those skilled in the art that the sequence of these actions may be modified without departing from the spirit and scope of the present invention. For example, as will be discussed later in this description, the order in which the processor tasks are copied from and written back into the shared memory 106 and the order in which the task table and task queue 282 are utilized, may be modified in order to achieve desirable results.

[0090] As discussed above, the main processing unit 102A is preferably utilized during an initialization phase of the system in order to place the system in a state in which the execution and management of the processor tasks 110 may be

handled by the sub-processing units 102. The sub-processing units 102 also preferably conduct an initialization routine in order to create the task table 280 and the task queue 282 in the first instance. These initialization processes are illustrated in the flow diagram of FIG. 11.

[0091] At action 350, the service processor (e.g., the main processing unit 102) evaluates the software application to be executed on the system and allocates a number of the sub-processing units 102 to execute the processor tasks 110. The process flow preferably advances to action 352 where the service processor evaluates the software application and allocates one or more portions of the shared memory 106 to receive the processor tasks 110. At action 354, the processor tasks 110 are loaded into the shared memory 106 in accordance with any memory allocation that was performed at action 352. At this stage of the initialization process, the service processor is preferably no longer involved in the maintenance and/or allocation of the processor tasks among the sub-processing units 102.

[0092] The process flow preferably advances to action 356, where the sub-processing units 102 initialize with one another to determine which SPU is to prepare the task table 280 and the task queue 282 in the first instance. At action 358, the sub-processing unit 102 that was charged to create the task table 280 and the task queue 282 prepares such information and stores same in the shared memory 106. By way of example, the initialization of the task table 280 and task queue 282 is preferably performed by having each SPU kernel execute an initial task. The program, init.c, reproduced below, is a preferred example of the initial task performed by each SPU:

```
#include <spurs.h>
#include "task_instance.h"

int
main()
```

```
        {
          spurs_beggin_init();

          if (spurs_get_spu_id() == 0) {
            spurs_create_task(melchior);
            spurs_create_task(balthasar);
            spurs_create_task(caspar);

            spurs_start_task(melchior);
            spurs_start_task(balthasar);
            spurs_start_task(caspar);
          }

          spurs_end_init();
          return 0;
        }
```

[0093]   In   this   program,   'melchior',   'balthasar'   and
'caspar' are the names of the very beginning tasks, which are
typical start up tasks.   All of the SPU kernels execute this
initial task, init.c, but only one SPU – the SPU with ID0 –
executes these tasks, as specified by the line of code:
if(spurs_get_spu_id ( ) = = ∅).   All the other SPUs, e.g.,
those with a different ID, wait at spurs_end_init().   Thus,
each of the SPU kernel executes the initial task and after
this initial task finishes, the SPU kernels just start looking
for next task as described herein.

[0094]   It is noted that the main processing unit 102,
acting as a service processor as discussed above, may
designate one or more of the processor tasks 110 as being
within a group.   This is preferably performed during the
initialization stage.   For example, two or more processor
tasks 110 may need to tightly communicate with one another
and, therefore, may be executed more efficiently if they are
grouped together in a task group.   An encryption program is an
example of an application that may contain processor tasks
that communicate tightly and would be more efficiently
executed if they were formed into one or more task groups.

[0095]   The processor task management features of the
present invention may be utilized to assist the main

processing unit 102A to off-load a device driver to a particular sub-processing unit 102 or group of sub-processing units 102. By way of example, a network interface, such as a gigabit Ethernet handler, may utilize up to 80% of CPU power. If the network interface is executed solely by the main processing unit 102A, then the main processing unit 102A may not be available to do other service-oriented processing tasks. Accordingly, it may be beneficial for a main processing unit 102A to off-load the network interface program to one or more of the sub-processing units 102. The main processing unit 102A may achieve this result by placing the processing tasks of the network interface into the shared memory 106 and allocating one or more sub-processing units 102 to execute same. In response, the SPUs may form a task table 280 and a task queue 282 suitable for managing and scheduling the execution of such processor tasks. Advantageously, the main processing unit 102A may therefore devote more CPU power to executing other tasks. The main processing unit 102A may also off-load other device drivers, such as a digital television device driver. Other device drivers that are good candidates for off-loading to SPUs are those that have heavy protocol stacks. For example, drivers for real-time high speed access devices, such as for a HDD recorder, can be advantageously off-loaded. Other examples of tasks that can be off-loaded include network packet encryption/description tasks used for Virtual Private Networks and multimedia over IP (e.g. VoIP) applications.

[0096] Reference is now made to FIG. 12, which is an example of a state diagram of the status of the processor tasks. The task states may be classified into five categories: the RUNNING state, the READY state, the BLOCKED state, the DORMANT state, and the NON-EXISTENT state. A processor task is in the RUNNING state when it is currently executing. Under some circumstances, a processor task may

remain in the RUNNING state even in non-task contexts, such as during interrupts. A processor task is in the READY state when the task is ready to be executed, but cannot be executed because one or more processor tasks with higher precedence are already being executed and a sub-processing unit is not available to take up the task. Once the priority level of the READY processor task is high enough among the pool of READY tasks in the shared memory 106, a sub-processing unit may take up the processor task and execute same. Thus, the state of a processor task may change from the READY state to the RUNNING state when the task is dispatched. Conversely, the task state of a RUNNING task may change to the READY state if such task is preempted or otherwise usurped during its execution. An example of the preemption of a processor task was discussed hereinabove in connection with one processor task yielding to another.

[0097] The BLOCKED state category may include the WAITING state, the SUSPENDED state, and the WAITING-SUSPENDED state. A processor task is in the WAITING state when the execution of such task is blocked due to the invocation of a service call specifying that certain conditions must be met before execution of the task continues. Thus, the state of a RUNNING task may change to the WAITING state upon the invocation of a service call. A processor task in the WAITING state may be released into the READY state when the specified conditions are met, thereby enabling the processing task to be taken up by a sub-processing unit 102 thereafter. A processor task may enter the SUSPENDED state from the RUNNING state when the task is forcibly halted (which the task itself may invoke). Similarly, a processor task in the READY may enter the SUSPENDED state through forced action. A SUSPENDED processor task may be resumed and enter the READY state when the forcible halting of such processor task is released. A processor task is in the WAITING-SUSPENDED state when the task

-27-

is both waiting for a condition to be met and when it has been forcibly suspended. Thus, a WAITING-SUSPENDED processor task may enter the WAITING state when the forcible suspension of the PROCESSOR task, where the processor task will wait for the condition to be satisfied.

[0098] A processor task is in the DORMANT state when the task has not been executed or has already finished its execution. A DORMANT processor task may enter the READY state under appropriate circumstances. The NON-EXISTENT state is a so-called virtual state where the task does not exist in the system, for example, because it has not yet been created or because it has already been deleted.

[0099] If a task that has been moved to the READY state has a higher precedence (or priority) than a task in the RUNNING state, the lower precedence task is preferably moved to the READY state and the higher priority task is preferably dispatched and moved to the RUNNING state. In this circumstance, the lower priority task has been preempted by the higher priority task.

[0100] Non-preemptive, priority-based task scheduling is conducted based on the priorities assigned to the processor tasks. If there are a number of processor tasks with the same priority, scheduling is conducted on a first-come, first-served (FCFS) basis. This task scheduling rule may be defined using a precedence between tasks based on task priorities. If a runable task exists, at most the same number of high precedence tasks as allocated sub-processing units 102 will be in the RUNNING state. The remainder of the runable tasks will be in the READY state. Among the tasks with different priorities, the task with the highest priority has higher precedence. Among tasks of the same priority, the processor task that entered the runable (running or ready) state earliest has the higher precedence. However, the precedence between tasks of the same priority may change do to

the invocation of some service calls. When a processor task is given precedence over other processor tasks, it is preferred that a dispatch occurs immediately and that the task is moved into the RUNNING state.

[0101] Reference is now made to FIGS. 13-14, which illustrate certain preemption features in accordance with certain aspects of the present invention. As discussed above, a processor task in the RUNNING state (e.g., task A) may be preempted or otherwise yield to another processor task in the READY state (e.g., task B). As illustrated in FIGS. 13 and 14, task A is being executed on the sub-processing unit 102 up to the point of the yield. At that point, the kernel of the SPU operates to copy task A back to the shared memory 106 (saving task A). Thereafter, task B is copied from the shared memory 106 to the local memory of the SPU (restoring task B). The SPU then executes task B. While this technique enjoys relatively high performance with reference to the amount of local memory utilization and high bandwidth, there is task execution latency from the point of the yield to the execution of task B that is not optimized.

[0102] With reference to FIGS. 15-16, an alternative approach is illustrated in accordance with further aspects of the present invention. In this scenario, task B may be copied from the shared memory 106 to the local memory of the sub-processing unit 102 prior to copying task A from the local memory to the shared memory 106. In this regard, the sub-processing unit 102 may execute task A, while at the same time taking steps to identify and retrieve task B from the shared memory 106. This may entail copying the task table 280 and the task queue 282 from the shared memory 106 to the local memory of the sub-processing unit 102A and using same to identify the next READY task, i.e., task B. At the yield point, the kernel of the sub-processing unit 102A copies task A from the local memory to the shared memory 106, which may

-29-

entail modifying the task table 280 and the task queue 282 as described hereinabove. Thereafter, the sub-processing unit 102 may take up the execution of task B. This technique significantly reduces the latency between the yield and the execution of task B as compared with the technique illustrated in FIGS. 13-14.

[0103] With reference to FIGS. 17 and 18, the latency between the yield and the execution of task B may be further reduced in accordance with one or more further aspects of the present invention. In particular, the sub-processing unit 102 may operate in a substantially similar way as discussed hereinabove with respect to FIGS. 15 and 16 up to the point of the yield. After the yield, however, the sub-processing unit 102 preferably initiates the execution of task B. At substantially the same time, the kernel of the sub-processing unit 102 preferably operates to copy task A from the local memory of the sub-processing unit 102 to the shared memory 106. Since task B is executed very shortly after the yield, the latency is significantly reduced as compared with the approaches illustrated in FIGS. 14-16.

[0104] In accordance with one or more further aspects of the present invention, the sub-processing units 102 may maintain multiple processor tasks in the local memory for execution. This is illustrated in FIG. 19. The local memory includes a plurality of pages and a page table in order to manage the execution of multiple processor tasks. The advantage of this approach is that latencies may be further reduced, although one drawback is that significantly more space within the local memory will be monopolized by process task execution.

[0105] Reference is now made to FIGS. 20-22, which illustrate processor task migration in accordance with one or more aspects of the present invention. These figures illustrate how a processor task, for example, task B may be migrated from one sub-processing unit SPU1 to another sub-processing unit

SPU2. The migration may be based on some condition, such as respective priority levels associated with the respective processor tasks. In accordance with some aspects of the invention, the migration of the processor task from one sub-processing unit to another may not be preemptive. In other words, the migration of the processor task may naturally flow as a result of priority conditions and timing, but are not based on some decision to cause the migration.

[0106] This non-preemptive migration may be illustrated by the following example. Assume that processor task B was selected from the shared memory 106 utilizing the task table, which dictates a priority order for the processor tasks that are ready to be executed. Task B is running on sub-processing unit SPU1. Similarly, it is assumed that processor task C was selected from the shared memory 106 in accordance with the task table and is running on sub-processing unit SP2. At the time that processor task B and processor task C were selected, it is assumed that processor task A of a higher priority level was not ready to be executed and, therefore, was not selected for execution. While processor task B and processor task C are running, however, it is assumed that processor task A becomes ready for execution.

[0107] With reference to FIG. 21, processor task B may yield sub-processing unit SP1. This yielding action by processor task B may occur due to the programmer having decided that a yield would be beneficial to the overall execution of the software application. In any case, sub-processing unit SP1 responds to the yield by writing processor task B back into the shared memory 106 and updating the task table. The sub-processing unit SP1 also accesses the task table to determine which of the plurality of processor tasks in the shared memory 106 should be copied and executed. In this example, processor task A is of a highest priority according to the task table and, therefore, the sub-processing unit SP1

copies processor tasks A from the shared memory 106 into its local memory for execution. At this point, sub-processing unit SP1 executes processor task A and sub-processing unit SP2 continues executing processor task C.

[0108] With further reference to FIG. 22, the processor task C may yield the sub-processing unit SP2 to another processor task. Again, the yield may be invoked through program instructions and/or conditions of the processor task C. In any case, the sub-processing unit SP2 writes the processor task C back into the shared memory 106 and updates the task table accordingly. The sub-processing unit SP2 also accesses the task table to determine which of the processor tasks that are ready to be executed should be copied. In this example, processor task B is ready to be executed and is of the highest priority among the plurality of processor tasks that are ready to be executed. Accordingly, the sub-processor unit SP2 copies the processor task B from the shared memory 106 into its local memory for execution.

[0109] A comparison of the processing conditions illustrated in FIG. 20 and the processing conditions illustrated in FIG. 22 reveals that processor task B has migrated from the sub-processing unit SPU1 to the sub-processing unit SP2.

[0110] Reference is now made to FIGS. 23-24, which illustrate preemptive multi-tasking aspects of the present invention. These aspects of the present invention provide that a processor task of lower priority running on one of the sub-processing units, such as the sub-processing unit SP2, may be preemptively replaced with a processor task of higher priority, such as processor task A. More particularly, processor task B may be running on the sub-processing unit SPU1, while processor task C may be running on the sub-processing unit SPU2 (FIG. 23). Thereafter, a higher priority task, task A, may become ready for execution. This

may occur due to some action by another sub-processing unit of the system.

[0111] For purposes of discussion, it is assumed that the sub-processing unit SPU1 changed the status of the processor task A to RUNNING, for example, as a result of executing processor task B. As a result, the sub-processing unit SPU1 preferably makes a determination as to whether the priority level of the processor task A is higher than any of the priority levels of the processor tasks running on the other sub-processing units. In this simplified case, the sub-processing unit SPU1 makes a determination as to whether the processor task A is of a higher priority level than the processor task C. If so, the sub-processing unit SPU1 at least initiates the replacement of the processor task C with the processor task A. In other words, the sub-processing unit SPU1 preferably causes the processor task C to yield the sub-processing unit SPU2 to the processor task A. In this regard, the kernel of the sub-processing unit SP1 may issue an interrupt to the kernel of the sub-processing unit SPU2. In response to the interrupt, the sub-processing unit SPU2 may write the processor task C back to the shared memory 106 and update the task table (FIG. 24). The sub-processing unit SPU2 may also copy the processor task A from the shared memory to its local memory for execution.

[0112] Reference is now made to FIGS. 25-26, which illustrate certain direct migration aspects of the present invention. These aspects provide that a processor task of higher priority running on one of the sub-processing units may be migrated to another of the sub-processing units running a processor task of lower priority. This migration may be in response to a direct interrupt received by the sub-processing unit running the higher priority processor task. With reference to FIG. 25, the sub-processing unit SPU1 may receive an interrupt that indicates that it must execute some other task. The

-33-

interrupt may also cause the sub-processing unit SPU1 to make a determination as to whether any of the other sub-processing units of the system are executing processing tasks of lower priority. If so, such sub-processing units may yield the execution of the processor task in favor of the higher priority processing task. More particularly, if the sub-processing unit SP1 determines that the sub-processing unit SP2 is running a processor task, such as processor task B, of a lower priority than the processor task A, then the kernel of the sub-processing unit SP1 preferably issues an interrupt to the kernel of the sub-processing unit SPU2. In response to the interrupt, the sub-processing unit SPU2 preferably writes the processor task B from its local memory back to the shared memory 106 and updates the task table. The sub-processing unit SPU2 also preferably copies (or moves) the processor task A from the local memory of the sub-processing unit SP1 to its local memory for execution.

[0113] FIG. 27 illustrates how a processing unit ("PU") can handle an interrupt in accordance with one aspect of the present invention. In a first step, the PU receives an interrupt. The PU then determines which sub processing unit — in this case, from the group of SPU0, SPU1 and SPU2 — has the lowest priority. Then, the PU sends the interrupt to the SPU with the lowest priority. In the case of FIG. 27, SPU2 has the lowest priority, so the PU would send an interrupt to the SPU2.

[0114] In accordance with one or more further aspects of the present invention, the interrupt from one of the sub-processing units to another of the sub-processing units may be handled in a number of ways. Reference is now made to FIG. 28, which illustrates how in one embodiment of the present invention, one sub-processing unit may be designated to manage interrupts to any of the other sub-processing units in the system. The designated sub-processing unit receives

all such task migration interrupts and either handles them itself or makes a determination as to whether to pass the interrupt on to another sub-processing unit. For example, if the interrupt is intended for the designated sub-processing unit, then it may handle the interrupt itself. Alternatively, if the interrupt is not intended for the designated sub-processing unit, then the designated sub-processing unit may send the interrupt to the sub-processing unit of the group that is executing a processor task of lowest priority.

[0115] FIG. 29 illustrates an alternative approach in which a distributed interrupt handling scheme may be employed. In accordance with this technique, respective interrupts are assigned to each sub-processing unit, for example, interrupt A may be assigned to sub-processing unit SPU0. Interrupts B, C may be assigned to the sub-processing unit SPU1, and interrupt D, E, F may be assigned to the sub-processing unit SPU2.

[0116] The discussion hereinabove with respect to FIGS. 23-26 entailed that the sub-processing units be capable of determining the priority levels of the processor tasks running on the other sub-processing units of the system. In accordance with one embodiment of the present invention, the sub-processing units may utilize a shared task priority table in determining the priority levels of the running processor tasks. The shared task priority table may be located in the shared memory and may include a plurality of entries for sub-processing unit identifiers and processor task priority identifiers. For example, the sub-processing unit identifiers may be numerical and/or alphanumerical codes unique to the sub-processing units. The processor task priority identifiers preferably indicate what priority level of the particular processor tasks being executed. Each entry in the shared task priority table preferably includes a pair: a sub-processing unit identifier and a priority identifier, indicative of a priority level of a given processor task running on an

associated sub-processing unit. Thus, a sub-processing unit seeking to determine the priority levels of the running processor tasks may access the shared task priority table to find a sub-processing unit running a lower priority processor task. Preferably, the sub-processing unit running a lowest priority processor task is identified for yielding to the higher priority level processor task.

[0117] Other embodiments of the present invention may provide that the sub-processing units utilize a shared variable that indicates which sub-processing unit is running a lowest priority processor task. The use of a shared variable is preferably achieved by way of atomic update processes such that accurate indications of priority levels are guaranteed. An alternative approach may utilize a serialized message that is transmitted from one sub-processing unit to another sub-processing unit in series. The message may be updated with the priority level and sub-processing unit identifier of lower priority level processor tasks.

[0118] With reference to FIG. 30, alternative embodiments of the present invention contemplate that the number of sub-processing units 208 allocated to execute the processor tasks 110 may be increased by joining a number of processor elements 200 together to provide enhanced processing power. For example, two or more processor elements 200A, 200B may be packaged or joined together, e.g., within one or more chip packages, to form a set of multi-processor units. This configuration may be referred to as a broadband engine (BE). The broadband engine 290 contains the two processor elements 200A, 200B, which are interconnected for data communication over a bus 212. An additional data bus 216 is preferably provided to permit communication between the processor elements 200A, 200B and the shared DRAM 214. One or more input/output (I/O) interfaces 202A and 202B and an external bus (not shown) provide communications between the broadband

engine 290 and any external elements. Each of the processor elements 200A and 200B of the broadband engine 290 perform processing of data and applications in a parallel and independent manner analogous to the parallel and independent processing of applications and data performed by the sub-processing elements 208 discussed hereinabove with respect to FIG. 3. In accordance with various aspects of the present invention, the BE may include a single processor element or multiple processor elements. Additionally, a BE may be formed from multiple BE's.

[0119] Reference is now made to FIG. 31. Here, the stand-alone multi-processing elements 208 or BEs 290 (sets of multi-processing elements) may be distributed among a plurality of products to form a multi-processing system 500. The elements or members (implemented as computer and/or computing devices) of the system 500 are preferably in communication over a network 504. The network 504 may be a local area network (LAN), a global network, such as the Internet, or any other computer network.

[0120] The members that are connected to the network 504 include, e.g., client computers 506, server computers 508, personal digital assistants (PDAs) 510, digital televisions (DTV) 512, and other wired or wireless computers and computing devices. For example, the client 506A may be a laptop computer constructed from one or more of the PEs 200 or other suitable multi-processing systems. The client 506B may be a desk-top computer (or set top box) constructed from one or more of the PEs 200 or other suitable multi-processing systems. Further, the server 506A may be a administrative entity (employing a database capability), which is also preferably constructed from one or more of the PEs 200. And so on.

[0121] Thus, the processing capabilities of the multi-processing system 500 may rely on a plurality of

processor elements 200 disposed locally (e.g., one product) or disposed remotely (e.g., in multiple products). In this regard, reference is made to FIG. 32, which is a block diagram of an overall computer network in accordance with one or more aspects of the present invention. Again, the PEs 200 and/or broadband engines 280 (made of multiple PEs) may be used to implement an overall distributed architecture for the computer system 500.

[0122] Since the servers 508 of the system 500 perform more processing of data and applications than the clients 506, the servers 508 contain more computing modules (e.g., PEs 200) then the clients 506. The PDAs 510, on the other hand, in this example perform the least amount of processing. Thus, the PDAs 510 contain the smallest number of PEs 200, such as a single PE 200. The DTVs 512 perform a level of processing that is substantially between that of the clients 506 and the servers 508. Thus, the DTVs 512 contain a number of processor elements between that of the clients 506 and the servers 508.

[0123] Further details concerning the distributed multi-processing system 500 will now be provided. The homogenous configuration for the system 500 facilitates adaptability, processing speed, and processing efficiency. Because each member of the system 500 performs processing using one or more (or some fraction) of the same computing module, e.g., processor element 200, the particular computer or computing device performing the processing of data and/or application is unimportant because the processing of such data and applications may be shared among the network's members. By uniquely identifying the software cells comprising the data and applications processed by the system 500, the processing results can be transmitted to the computer or computing device requesting the processing regardless of where this processing occurred. Because the modules performing this processing have a common structure and employ a common instruction set

architecture, the computational burdens of an added layer of software to achieve compatibility among the processors is avoided. This architecture and programming model facilitates the processing speed necessary to execute, e.g., real-time, multimedia applications.

[0124] To take further advantage of the processing speeds and efficiencies facilitated by the system 500, the data and applications processed by this system may be packaged into uniquely identified, uniformly formatted software cells 502. Each software cell 502 contains, or can contain, both applications and data. Each software cell also contains an ID to globally identify the cell throughout the network 504 and the system 500. This uniformity of structure for the software cells, and the software cells unique identification throughout the network, facilitates the processing of applications and data on any computer or computing device of the network 504. For example, a client 506 may formulate a software cell 502 but, because of the limited processing capabilities of the client 506, transmit the software cell 502 to a server 508 for processing. Software cells 502 can migrate, therefore, throughout the network 504 for processing on the basis of the availability of processing resources on the network 504.

[0125] The homogenous structure of processors and software cells 502 of the system 500 also avoids many of the problems of today's heterogeneous networks. For example, inefficient programming modules which seek to permit processing of applications on any ISA using any instruction set, e.g., virtual machines such as the Java virtual machine, are avoided. The system 500, therefore, can implement broadband processing far more effectively and efficiently than conventional networks.

[0126] Co-assigned United States Patent Application Serial No. _____, entitled METHODS AND APPARATUS FOR TASK MANAGEMENT IN A MULTI-PROCESSOR SYSTEM, having attorney docket no.

SCEI 1.0-186, and being filed in the United States Patent and Trademark Office on the same date as the present application, is hereby incorporated by reference in its entirety.

[0127] Although the invention herein has been described with reference to particular embodiments, it is to be understood that these embodiments are merely illustrative of the principles and applications of the present invention. It is therefore to be understood that numerous modifications may be made to the illustrative embodiments and that other arrangements may be devised without departing from the spirit and scope of the present invention as defined by the appended claims.